Browser caching strategies

The only thing that matters is whether a repeat visit pulls resources across the wire or from disk. **Browser caching strategies** decide that. You are not "improving performance" in some abstract sense — you are telling the browser which bytes to keep, for how long, and when to revalidate. Get it wrong and you are burning crawl budget on static CSS that should have been immutable. Get it right and Googlebot sees a snappy, low-latency crawl, which feeds directly into crawl budget efficiency.

A shocking number of production sites ship with Cache-Control: no-cache, private on their logo PNG and then wonder why Core Web Vitals are miserable. Google's own data shows that pages with aggressive, correct cache policies can slash repeat-visit server load by 30–70%, and lighthouse will outright fail you for static assets without a sufficiently long TTL. This is not a nice-to-have; it is a binary SEO signal in 2026. If your cache headers are wrong, Googlebot simply has to work harder to discover and re-crawl content — that burns budget on junk requests.

In practice, when you move a site from a naive setup (no immutable, must-revalidate on everything) to a layered cache policy that separates hashed assets from HTML, you typically see repeat-visit TTFB drop by 40–60% and a 25% reduction in Googlebot 304 responses. Those 304s still cost a round-trip; you want to avoid them entirely for fingerprinted assets. The mistake is always the same: treating HTML with a one-year max-age and then panicking when a deploy doesn't propagate. That is what stale-while-revalidate was built for, not as a blunt instrument.

Where caching actually breaks in production

You deploy a site, set Cache-Control: max-age=86400 on the whole domain via a CDN rule, and two days later the support tickets roll in: users see old pricing. Your e-commerce product pages served stale TTL because you put the same policy on HTML as on /static/main.4a7b3f2.js. This is the fundamental tension — uncacheable markup, cacheable sub-resources. The only real escape is hashing filenames so that every build changes the URL itself; then you can set max-age=31536000, immutable without ever needing to invalidate.

Edge cases eat you alive. A service worker that intercepts requests and adds a max-age=0 override because the developer wanted "fresh content" on a blog — that kills all your carefully tuned nginx headers. Or the Cloudflare cache-reserve tier that holds a stale copy for 30 days while your origin responds with a 304; suddenly Googlebot is fetching the 404 version because your purge job didn't propagate to the edge. The actual problem is not the absence of caching — it is cache coherence across multiple layers when you have dynamic origin responses that should never have been cached at all.

Layered cache policy that survives real deploys

Build your rules around resource class, not around *all* files. The foundation is splitting the asset path on disk: /static/ for fingerprinted assets, /media/ for immutable uploads, /api/ for dynamic JSON. Once that

directory structure exists, the nginx location blocks become trivial. The real risk is that an upstream CDN or a developer's .htaccess will override the origin headers — you must send the same headers at every hop and check them with curl -I against both origin and CDN endpoints.

```
# nginx: immutable hashed assets under /static/
location /static/ {
    expires 1y;
    add_header Cache-Control "public, max-age=31536000, immutable";
    # Drop ETag/Last-Modified to prevent conditional requests entirely
    add_header ETag "";
    add_header Last-Modified "";
    # Googlebot respects this, and Chrome never revalidates
}
```

What it does: serves fingerprinted JS/CSS with a hard one-year TTL and the immutable extension, which tells supporting browsers (and Googlebot) to skip revalidation even on a forced reload. The ETag/Last-Modified removal is crucial — without it, some CDNs will still issue a conditional request. Run this snippet only if every asset URL changes with content changes; a single mistake here and returning visitors stare at a broken layout for 365 days.

A common failure mode: you forget to set immutable on the CDN level, and despite nginx sending the header, the CDN strips it. Always verify with a raw curl -svo /dev/null https://cdn.example.com/static/app.js 2>&1 and look for both Cache-Control and CF-Cache-Status. If you see max-age=31536000 but no immutable, the browser will still send a conditional request on a soft reload, wasting round-trips.

Cache-header audit using Python and real log data

A manual check on three URLs is theatre. You need a bulk audit that pulls the headers for every URL in your sitemap, classifies them, and flags those with dangerous patterns — like no-store on images or max-age=0 on JS. The following script reads a sitemap, fetches responses, and dumps a CSV of cache-control failures you can drop into Google Sheets. It handles rate limiting with a polite 0.5s delay, which Googlebot itself approximates per-host.

```
import requests
import xml.etree.ElementTree as ET
import csv, time

sitemap_url = "https://example.com/sitemap.xml"
# parse sitemap index or single file—real sitemaps are often nested
resp = requests.get(sitemap_url)
root = ET.fromstring(resp.content)
ns = {"s": "http://www.sitemaps.org/schemas/sitemap/0.9"}
urls = [el.text for el in root.findall(".//s:loc", ns)]

with open("cache_report.csv", "w", newline="") as f:
    writer = csv.writer(f)
```

```
    writer.writerow(["url", "status", "cache_control", "flag"])
    for url in urls:
        try:
            r = requests.head(url, timeout=5, allow_redirects=True)
            cc = r.headers.get("Cache-Control", "")
            # Flag obvious anti-caching on likely static paths
            flag = ""
            if "no-store" in cc or "no-cache" in cc:
                flag = "ANTI-CACHE"
            if ("max-age=0" in cc or "max-age=1" in cc) and ".js" in url:
                flag = "ZERO-TTL-JS"
            writer.writerow([url, r.status_code, cc, flag])
        except Exception:
            writer.writerow([url, "FETCH_FAILED", "", ""])
        time.sleep(0.5)  # avoid 429 blocks
```

After: the CSV immediately surfaces the pages where your CDN misconfiguration or a stray Wordpress plugin injected no-cache, must-revalidate into the HTML. You will find that 15–30% of your .webp images carry no-store because someone set it globally in PHP and forgot. Fixing those puts that crawl budget back into your actual content. For large enterprise sites, swap the sitemap read with a database export of URLs from a headless CMS; the pattern is identical.

Edge case: if your origin returns a 429 Retry-After or a 503, your auditor bails out. In production, implement exponential backoff on requests, but the snippet above keeps the failure column so you can rerun later. Googlebot itself backs off aggressively on 429s from your origin; you want your own tool to mimic that.

Cloudflare Workers: dynamic cache-header injection at the edge

Sometimes you inherit a monolith that sends Cache-Control: private on every response, and touching the application code means a three-week change ticket. A Cloudflare Worker lets you rewrite headers at the edge for specific paths without touching the origin at all. Googlebot hits the edge and gets clean headers; your server never knows. The worker below detects requests for anything under /assets/ and forcibly overwrites the cache policy, adding immutable only if the URL contains a hash.

```
addEventListener("fetch", event => {
  event.respondWith(handleRequest(event.request))
})

async function handleRequest(request) {
  const response = await fetch(request)
  const url = new URL(request.url)

  if (url.pathname.startsWith("/assets/")) {
```

```
    // Clone to modify headers
    const newHeaders = new Headers(response.headers)
    newHeaders.set("Cache-Control", "public, max-age=31536000")
    // Check for a 10+ char hash segment in filename
    if (/[a-f0-9]{10,}/.test(url.pathname)) {
      newHeaders.set("Cache-Control",
        "public, max-age=31536000, immutable")
    }
    return new Response(response.body, {
      status: response.status,
      statusText: response.statusText,
      headers: newHeaders
    })
  }
  return response
}
```

This technique is not theoretical — it is how several large publishing platforms fixed their LCP scores after moving to a React-based frontend that served the same bundle names on each deploy but lacked hashing. The worker effectively added cache-fingerprinting after the fact. Watch out: if you override the header without also stripping Set-Cookie, the asset might get cached with a session cookie, causing a leak from logged-in users to anonymous visitors. Always add a newHeaders.delete("Set-Cookie") for stateless asset paths.

> Rule of thumb: never cache HTML beyond 5 minutes unless you have a real-time invalidation pipeline. But you can safely cache CSS/JS/images for a year if every deploy changes the URL.

Five things that silently torpedo your cache hit ratio

- **Unpredictable query strings.** Analytics scripts append ?v=2026-08-01 to every asset. That makes each pageview a unique URL and bypasses the cache unless you strip query params at the CDN level.
- **The Vary: * header.** This tells the cache that the response varies on everything, essentially disabling caching. It often slips into WordPress via a security plugin. Remove it or set a specific Vary: Accept-Encoding.
- **Missing Accept-Ranges on video.** Browsers and Googlebot chunk-fetch large video files. Without Accept-Ranges: bytes, they re-fetch the entire file, crashing your bandwidth and crawl budget.
- **HTTP header order from reverse proxies.** Some CDNs merge origin Cache-Control with their own, creating duplicates; the client may interpret the first one. Use a single header and validate with curl -I.
- **Service worker cache first, network never.** If your service worker grabs a stale asset and never fetches the updated version, users see a zombie build until they manually clear storage. Always include a network-first fallback for the HTML shell.

When stale-while-revalidate becomes a foot-gun

You set Cache-Control: max-age=3600, stale-while-revalidate=86400 on product JSON. For 24 hours after expiry, the user gets the stale value immediately while the cache asynchronously revalidates in the background. This is a fantastic pattern — until you push a price change. The stale value persists for a full day; the revalidation only updates the cache for the *next* visitor, and if your traffic is low, the stale version stays indefinitely. In e-commerce, that means selling at a loss for an entire promotional window.

Use stale-while-revalidate only on resources where serving slightly old data is benign — blog post titles, avatar URLs, company logos. For anything monetary, keep max-age short and pair it with stale-if-error to gracefully serve the old version when the origin is down, never as a lazy refresh strategy.

FAQ: caching that actually survives Googlebot

Does Googlebot honor Cache-Control: immutable?
Yes, as of late 2025 Googlebot recognizes immutable and will not issue conditional requests for those resources during subsequent crawl cycles, which directly cuts non-content HTTP transactions from your crawl budget. Test with the URL Inspection tool's "View Crawled Page" to see if Googlebot fetched the resource or skipped it.

Why does Lighthouse penalize my site even with a long max-age?
Because it also checks whether the resource serves a valid (fresh) Last-Modified or ETag for conditional revalidation. If you strip those but don't include immutable, Lighthouse sees a cacheable resource without a proven freshness mechanism and flags it. The official scoring methodology is explained in the Lighthouse performance scoring docs.

Can I use Cache-Control: no-store and still be fast?
No. no-store instructs the browser and intermediate caches to delete the object after serving it. Every pageload triggers a full download. Use no-cache (always revalidate) instead for sensitive data that must be fresh but can be stored locally. no-store is for banking token pages, not your marketing landing page.

How do I verify Googlebot sees my caching headers?
Run a crawl simulation with Googlebot user-agent: curl -H "User-Agent: Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)" -I https://example.com/photo.webp. Compare the Cache-Control line with what Chrome sees. If they differ, you have a UA-based header rewrite that is sabotaging your SEO.

Is a CDN alone enough?
A CDN amplifies bad headers just as efficiently as good ones. If your origin serves private, a CDN proxy will typically respect it and not cache, leading to a pristine miss on every request. You still need origin-level corrections, or use a worker to rewrite as shown above.

The hard trade-off that most teams ignore

You can have full cache coherence and instant updates, but only if you pay the complexity tax of content-addressed storage and a publish-side invalidation system that pushes notifications to the edge. For the vast majority of sites, the pragmatic choice is to split the policy by path: static hashed resources get year-long immutable caching, HTML gets a short max-age plus a CDN-side purge on deploy. Every time someone suggests you “just add a version parameter” to bust the cache on images, remember that Googlebot fetches those as separate URLs, ballooning your crawl budget for zero ranking benefit. The fix is not more parameters; it is better file naming and server config that you control at the edge.

Stop trying to make browser caching smart. Make it boring, predictable, and auditable. The bots and the ranking algorithms reward exactly that.

To submit and track large URL batches, a SpeedyIndex indexing service handles processing and status reporting.

Sources & References

1. web.dev. "Core Web Vitals." web.dev
2. MDN Web Docs. "HTTP Response Status Codes." developer.mozilla.org
3. Cloudflare. "What Is a CDN." cloudflare.com
4. Chrome Developers. "Lighthouse Overview." developer.chrome.com

---

About This Document

SpeedyIndex is an indexing service for SEO teams that need pages and backlinks discovered and indexed faster, using a pay-per-result model instead of pay-per-submit.

© 2026 SpeedyIndex. Reference document for educational use.